A Confluence Checker for Constraint Handling Rules with Persistent Constraints

Frank Richter, Daniel Gall, and Thom Frühwirth

Institute of Software Engineering and Programming Languages,
Ulm University, Germany

Abstract. In the abstract operational semantics of Constraint Handling Rules (CHR), propagation rules, i.e. rules that only add information, can be applied again and again. This trivial non-termination is typically avoided by a propagation history. A more declarative approach are persistent constraints. Constraints that are introduced by propagation rules are made persistent and cannot be removed. Now a propagation rule is only applied, if its derived constraints are not already persistent.
The operational semantics with persistent constraints $\omega_!$ differs substantially from other operational semantics, hence the standard confluence test cannot be applied. In this paper, a confluence test for $\omega_!$ is presented. Since $\omega_!$ breaks monotonicity of CHR, a weaker property is established that is shown to suffice for a decidable confluence criterion for terminating $\omega_!$ programs. The confluence test is implemented using a source to source transformation.

Keywords: Constraint Handling Rules, constraint programming, persistent constraints, confluence, propagation rules, source to source transformation

1 Introduction

Constraint Handling Rules (CHR) [3] is a declarative, multiset- and rule-based programming language. There exist several operational semantics.

The simplest and most basic operational semantics for CHR is the very abstract semantics. Its behavior is close to the logical reading of the rules. This leads to the problem of trivial non-termination with rules that do not remove any constraints and so are applicable any number of times. This class of rules is known as propagation rules. The way most operational semantics avoid this trivial non-termination problem is by adding a token store. This token store is used to ensure that a propagation rule is only applied once with the same constellation of constraints.

Such a token store can cause states with the same logical reading to exhibit a different operational behavior. It also hinders effective concurrent execution of CHR programs, since it needs to be distributed adequately [1].

The operational semantics with persistent constraints, denoted as $\omega_!$, adds a second constraint store for so called persistent constraints. Those are constraints

that represent any number of these constraints and cannot be removed. This allows it to stay close to the abstract semantics and avoid trivial non-termination without a token store, but the transition rules lose the monotonicity property, which is used in several proofs. CHR in general offers powerful program analyses and is suitable for concurrent execution [1]. Since $\omega_!$ avoids a token store, its concurrent execution is not compromised. For a program to be easily used in a concurrent execution the property of confluence is in general important. It implies that the order in which the rules are executed does not influence the result.

The contribution to this topic presented in our work is:

- The introduction of a modified state equivalence definition for $\omega_!$ to correct a flaw in the original definition. Additionally, a criterion for state equivalence according to the new definition is introduced. (Section 3)
- A confluence test for programs that terminate in $\omega_!$. (Section 4)
- An implementation of $\omega_!$ as source to source transformation. This transformation is based on the transformation presented in [2] and is realized in SWI Prolog. (Section 5.1)
- A tool that can check terminating programs for confluence in $\omega_!$. The checker has an extension that offers support for more built-ins. This tool is a modification and extension of the confluence checker for the abstract semantics of CHR [6]. (Section 5)

The paper starts with a short introduction to CHR and the introduction of $\omega_!$ in the preliminaries section. This is followed by a section with the extended state equivalence definition. The following section uses this definition to present a confluence test for $\omega_!$. The next section builds on this by introducing a tool that can check programs for confluence in $\omega_!$ with the help of a source to source implementation for $\omega_!$

2 Preliminaries

CHR is a rule based programming language, that needs a host language to provide support for built-in predicates. It consists of three different kind of rules. CHR has different operational semantics. This section starts by presenting the syntax of CHR. The very abstract semantics are introduced. Finally, the idea for the persistent semantics together with their definition is presented as they are originally introduced by [1].

2.1 Syntax

A CHR Program consists of a finite set of rules of the form $r \ @ \ H_k \backslash H_r \ \Leftrightarrow \ C|B$. A rule has an optional name r. There are built-in and CHR constraints of the form $c(t_1, ..., t_n)$, where c is a constant symbol, n is the arity and $t_1 ... t_n$ are first-order terms. Reasoning on built-in constraints can be done through

a satisfaction-complete and decidable constraint theory CT, while CHR constraints are simply user defined constraints [2]. Each type of rule has a head that may not be empty and consists of CHR constraints, a guard C that may be empty and consists of built-in constraints and a body B that may not be empty. The body can consist of built-in constraints as well as CHR constraints. H_k is the kept head and H_r the removed head. In simplification rules the kept head is empty, while in propagation rules the removed head is empty and $\Rightarrow$ is used instead of $\Leftarrow$. If neither head is empty it is a simpagation rule [3, p. 54].

2.2 Very Abstract Semantics ω_{va}

The very abstract operational semantics of CHR is given by a nondeterministic state transition system [3, p. 55].

Definition 1 (State). *A state is a conjunction of built-in and CHR constraints. An initial state (initial goal) is an arbitrary state and a final state is one where no more transitions are possible [3, p. 56].*

For the transitions, rules are used in head normal form (HNF). This means that each argument of a head constraint is a unique variable. A rule can be represented in HNF by replacing each of its head arguments t_i with a new variable V_i and adding the equation $V_i = t_i$ to the guard of the rule. The built-in $= /2$ for syntactic equivalence must be provided by CT. A transition represents a rule application according to the following transition relation of ω_{va}:

Apply

$$(H_k \wedge H_r \wedge C) \mapsto_r (H_k \wedge C \wedge B \wedge G)$$

if there is an instance with new local variables $\overline{x}$ of a rule named r in P.

$$r \ @ \ H_k \backslash H_r \Leftrightarrow G | B \text{ and } CT \models \forall (C \rightarrow \exists \overline{x} G)$$

The upper-case letters H_k, H_r, G, B and C represent conjunctions of constraints that can be empty. If H_k and H_r are present in the constraint store and G holds, the rule is applicable and the CHR constraints H_k are kept while the CHR constraints H_r are removed. The resulting state additionally consists of the guard G and the body B [3, p. 56].

This transition system is nondeterministic, because if several rules are applicable one is chosen nondeterministically and this choice cannot be undone [3, p. 56].

Since a rule is always applicable if the head constraints are present and the guard is satisfied an applicable propagation rule stays applicable after any number of applications. This causes the aforementioned trivial non-termination.

2.3 Operational Semantics with Persistent Constraints $\omega_!$

The operational semantics for $\omega_!$ is based on three basic ideas:

1. Propagation rules in ω_{va} cause trivial non-termination, since given the corresponding head constraints are present in the constraint store the body can be generated any number of times. To avoid this kind of trivial non-termination a second constraint store is introduced in which those body constraints are

added. Constraints in this store are a finite representation of a very large, though unspecified number of identical constraints, so called persistent constraints. To differentiate between persistent and non-persistent constraints, non-persistent constraints are called linear constraints [1].
2. If the removed head of a rule in ω_{va} consists entirely of constraints that can be generated any number of time, the body of such a rule can also be generated any number of times given the constraints of the kept head are also present. To account for those indirect consequences of propagation rules, a rule's body is introduced as persistent constraints, if its removed head is completely matched with persistent constraints [1].
3. Several occurrences of a persistent constraint are considered idempotent, since a persistent constraint represents an arbitrary number of identical constraints. For the execution model transitions are only supposed to happen if the post-transition state is not equivalent to the pre-transition state. This irreflexible transition system avoids trivial non-termination [1].

Definition 2 gives the definition for $\omega_!$ states [1].

Definition 2. *($\omega_!$-State).*
A $\omega_!$-state is a tuple of the form $\langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle$, where $\mathbb{L}$ and $\mathbb{P}$ are multisets of CHR constraints called the linear (CHR) store and the persistent (CHR) store, respectively. $\mathbb{B}$ is a conjunction of built-in constraints and $\mathbb{V}$ is a set of variables. The first state in a program execution is called initial state and can be any valid $\omega_!$ state.

Definition 3 defines the notion of local and strictly local variables, which is needed in definition 4 [1].

Definition 3. *(Local and strictly local variables).*
Let $\sigma = \langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle$ be an $\omega_!$ state. Then the variables occurring in $\mathbb{B}$ or in $\mathbb{L}$ or in $\mathbb{P}$ but not in $\mathbb{V}$ are called the local variables of σ. While the variables occurring in $\mathbb{B}$ but not in $\mathbb{L}$, $\mathbb{P}$ and $\mathbb{V}$ are called the strictly local variables of σ.

Definition 4 presents the definition of state equivalence and is based on the definition of state equivalence for ω_{va} given in [8] which has been extended by condition 5 to handle idempotence of persistent constraints [1].

Definition 4 (State Equivalence). *Equivalence between $\omega_!$ states is the smallest equivalence relation $\equiv_!$ over $\omega_!$ states that satisfies the following conditions [1]:*

1. *(Equality as Substitution)*
   Let X be a variable, t be a term and $\doteq$ the syntactical equality relation.
$$\langle \mathbb{L}, \mathbb{P}, X \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle \equiv_! \langle \mathbb{L}[X/t], \mathbb{P}[X/t], X \doteq t \wedge \mathbb{B}, \mathbb{V} \rangle$$
2. *(Transformation of the Constraint Store)*
   If $CT \models \exists \overline{s}.\mathbb{B} \leftrightarrow \exists \overline{s}'.\mathbb{B}'$ where $\overline{s}, \overline{s}'$ are the strictly local variables of $\mathbb{B}, \mathbb{B}'$ respectively, then:
$$\langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle \equiv_! \langle \mathbb{L}, \mathbb{P}, \mathbb{B}', \mathbb{V} \rangle$$

3. *(Omission of Non-Occurring Global Variables)*
   If X is a variable that does not occur in $\mathbb{L}$,$\mathbb{P}$ or $\mathbb{B}$ then:

$$\langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \{X\} \cup \mathbb{V} \rangle \equiv_! \langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle$$

4. *(Equivalence of Failed States)*

$$\langle \mathbb{L}, \mathbb{P}, \bot, \mathbb{V} \rangle \equiv_! \langle \mathbb{L}', \mathbb{P}', \bot, \mathbb{V} \rangle$$

5. *(Contraction of Persistent Constraints)*

$$\langle \mathbb{L}, P \uplus P \uplus \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle \equiv_! \langle \mathbb{L}, P \uplus \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle$$

Based on the state equivalence definition a rewrite system is defined over equivalence classes of states with $[G] := \{G' \mid G \equiv_! G'\}$. The fact that body constraints can be introduced as either persistent or linear constraints leads to two distinct transition rules. The post-transition state τ needs to be different from the pre-transition state σ. This means the transition relation is irreflexive. This definition is only valid for so called range restricted programs. Those are programs where no rule introduces free variables in the guard or body that are not also present in the head of the rule [1][2].

Definition 5. *($\omega_!$-Transitions)*
For a range restricted CHR program P, the state transition system $(\Sigma_!/\equiv_!, \mapsto_!)$ is defined as follows.
ApplyLinear:

$$\frac{r \ @ \ (H_1^l \uplus H_1^p) \setminus (H_2^l \uplus H_2^p) \Leftrightarrow G \mid B_c, B_b \qquad H_2^l \neq \emptyset \quad [\sigma] \neq [\tau]}{\begin{array}{c}\sigma = [\langle H_1^l \uplus H_2^l \uplus \mathbb{L}, H_1^p \uplus H_2^p \uplus \mathbb{P}, G \wedge \mathbb{B}, \mathbb{V} \rangle] \\ \mapsto_!^r [\langle H_1^l \uplus B_c \uplus \mathbb{L}, H_1^p \uplus H_2^p \uplus \mathbb{P}, G \wedge \mathbb{B} \wedge B_b, \mathbb{V} \rangle] = \tau\end{array}}$$

ApplyPersistent:

$$\frac{r \ @ \ (H_1^l \uplus H_1^p) \setminus H_2^p \Leftrightarrow G \mid B_c, B_b \qquad [\sigma] \neq [\tau]}{\begin{array}{c}\sigma = [\langle H_1^l \uplus \mathbb{L}, H_1^p \uplus H_2^p \uplus \mathbb{P}, G \wedge \mathbb{B}, \mathbb{V} \rangle] \\ \mapsto_!^r [\langle H_1^l \uplus \mathbb{L}, H_1^p \uplus H_2^p \uplus B_c \uplus \mathbb{P}, G \wedge \mathbb{B} \wedge B_b, \mathbb{V} \rangle] = \tau\end{array}}$$

where B_c are the CHR constraints and B_b are the built-in constraints of the body of a rule.

In cases where r is clear from the context or not important $\mapsto_!$ is used instead of $\mapsto_!^r$. With $\mapsto_!^*$ the reflexive-transitive closure of $\mapsto_!$ is denoted [2].

Example 1. (Transitive Hull)[1]
Consider the following CHR program for computing the transitive hull of a graph represented by edge constraints e/2:

$$t \ @ \ e(X,Y), e(Y,Z) \ \Rightarrow \ e(X,Z)$$

Called with $e(1,2), e(2,1)$ only four transitions are applied where $e(1,1), e(1,2), e(2,2)$ and $e(2,1)$ are added to the persistent store. No further transitions are possible since all resulting states would be equivalent.

For $\omega_!$ this program terminates for all possible inputs [1].

3 Extended State Equivalence Definition for $\omega_!$

In definition 4 state equivalence for $\omega_!$ is presented like it is introduced in [1]. In this definition the occurrence of linear constraints that are also present as persistent constraints has influence on state equivalence.

Example 2. Consider the following program:

$$\begin{aligned} a &\Leftrightarrow c. & a &\Leftrightarrow d. \\ b &\Rightarrow c. & b &\Rightarrow d. \end{aligned}$$

In ω_{va} this program does not terminate if called with a, b due to trivial non-termination. The constraint a can fire one of the two simplification rules which then lead to two different states. These states can fire propagation rules in a way that the resulting states are equivalent.

In $\omega_!$ the resulting final states of an execution with the initial state $\langle\{a,b\},\emptyset,\top,\emptyset\rangle$ is $\langle\{b,c\},\{c,d\},\top,\emptyset\rangle$ or $\langle\{b,d\},\{c,d\},\top,\emptyset\rangle$. Those two states are not equivalent, even so no rule can be constructed where the head and guard can only be matched by only one of the two states.

Our work introduces definition 6 as extension to definition 4. It adds one condition, which is based on the idea that if any number of a constraint is present adding more does not make a difference. This captures the nature of persistent constraints more accurately than the original definition. This behavior is not represented in the state equivalence definition of $\omega_!$ so far.

Definition 6. *Definition 4 is extended by the following axiom:*
(Contraction of Linear and Persistent Constraints)

$$6.\ \langle P \uplus \mathbb{L}, P \uplus \mathbb{P}, \mathbb{B}, \mathbb{V}\rangle \equiv_! \langle \mathbb{L}, P \uplus \mathbb{P}, \mathbb{B}, \mathbb{V}\rangle$$

Definition 6 gives an axiomatic definition for $\equiv_!$. It is difficult to show that something is not equal with an axiomatic definition. Definition 7 presents the $\bowtie$ relation [7] that is needed for Theorem 1 which gives a decidable criterion for $\equiv_!$. It is based on the criterion in [7] but takes definition 6 into account.

Definition 7 ($\bowtie$). *The relation $\bowtie$ over multisets of constraints is defined as*

$$\mathbb{G} \bowtie \mathbb{G}' \text{ if and only if } (\forall c \in \mathbb{G}.\exists c' \in \mathbb{G}'.c = c') \wedge (\forall c' \in \mathbb{G}'.\exists c \in \mathbb{G}.c = c')$$

Theorem 1 (Criterion for $\equiv_!$). *Let $\sigma = \langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V}\rangle$, $\sigma' = \langle \mathbb{L}', \mathbb{P}', \mathbb{B}', \mathbb{V}\rangle$ be $\omega_!$ states with local variables $\bar{y}, \bar{y}'$ that have been renamed apart. $\sigma \equiv_! \sigma'$ iff*

$$\begin{aligned} CT \models\ & \forall(\mathbb{B} \mapsto \exists \bar{y}'.((((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B}')) \wedge \\ & \forall(\mathbb{B}' \mapsto \exists \bar{y}.((((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B})) \end{aligned}$$

where $\triangle$ is the symmetric difference.

Proof. '$\Leftarrow$': Let σ and σ' be two $\omega_!$ states with $\sigma = \langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V}\rangle$, $\sigma' = \langle \mathbb{L}', \mathbb{P}', \mathbb{B}', \mathbb{V}\rangle$ with local variables $\bar{y}, \bar{y}'$ that have been renamed apart and

$$CT \models \forall(\mathbb{B} \mapsto \exists \bar{y}'.((((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B}')) \wedge$$
$$\forall(\mathbb{B}' \mapsto \exists \bar{y}.((((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B}))$$

If $CT \models \neg\exists(((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}')$, then $CT \models \mathbb{B} = \mathbb{B}' = \bot$ so that definition 4 condition 4 proves $\sigma \equiv_! \sigma'$.

If a matching for $(((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}')$ does exist it follows from $\forall(\mathbb{B} \mapsto \exists \bar{y}'.((((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B}') \wedge$ by definition 4 condition 2 that: $\sigma = \langle \mathbb{L}, \mathbb{P}, (((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B} \wedge \mathbb{B}', \mathbb{V} \rangle$
Definition 4 condition 1 and definition 6 lead to:

$$\sigma = \langle (\mathbb{L} \cap \mathbb{L}'), \mathbb{P}, (((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B} \wedge \mathbb{B}', \mathbb{V} \rangle$$

Definition 4 condition 1 and 5 lead to:

$$\sigma = \langle (\mathbb{L} \cap \mathbb{L}'), \mathbb{P}'', (((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B} \wedge \mathbb{B}', \mathbb{V} \rangle$$

where P'' equals P' modulo multiplicities. Definition 4 condition 5 and definition 6 now lead to: $\sigma = \langle \mathbb{L}', \mathbb{P}', (((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B} \wedge \mathbb{B}', \mathbb{V} \rangle$
From $\forall(\mathbb{B}' \mapsto \exists \bar{y}.((((\mathbb{L} \triangle \mathbb{L}') \uplus \mathbb{P}) \bowtie \mathbb{P}) \wedge (\mathbb{P} \bowtie \mathbb{P}') \wedge \mathbb{B})$ follows by definition 4 that: $\sigma = \langle \mathbb{L}', \mathbb{P}', \mathbb{B}', \mathbb{V} \rangle = \sigma'$

'$\Rightarrow$': To prove the forward direction the compliance of the conditions 1 to 5 from definition 4 and the condition form definition 6 need to be shown. For condition 1 to 4 of definition 4, compliance is analogous to [7][p.39f] and for condition 5 of definition 4 compliance is analogous to [7][p.47] hence only Definiton 6 is considered: Let $\sigma = \langle P \uplus \mathbb{L}, P \uplus \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle$, $\sigma' = \langle \mathbb{L}, P \uplus \mathbb{P}, \mathbb{B}, \mathbb{V} \rangle \in \Sigma_!$ with local variables $\bar{y}, \bar{y}'$. As $(((P \uplus \mathbb{L}) \triangle \mathbb{L}) \uplus \mathbb{P}) \bowtie (P \uplus \mathbb{P})) = ((P \uplus \mathbb{P}) \bowtie (P \uplus \mathbb{P}))$, the following is a tautology:

$$CT \models \forall(\mathbb{B} \mapsto \exists \bar{y}'.(((((P \uplus \mathbb{L}) \triangle \mathbb{L}) \uplus \mathbb{P}) \bowtie (P \uplus \mathbb{P})) \wedge ((P \uplus \mathbb{P}) \bowtie (P \uplus \mathbb{P})) \wedge \mathbb{B}) \wedge$$
$$\forall(\mathbb{B} \mapsto \exists \bar{y}.(((((P \uplus \mathbb{L}) \triangle \mathbb{L}) \uplus \mathbb{P}) \bowtie (P \uplus \mathbb{P})) \wedge ((P \uplus \mathbb{P}) \bowtie (P \uplus \mathbb{P})) \wedge \mathbb{B}))$$

□

Definition 8 introduces the merge operator $\diamond$ for merging $\omega_!$ states [7]. It is a technical definition needed for Lemma 3.

Definition 8 (Merge Operator $\diamond$). *Let* $\sigma_1 = \langle \mathbb{L}_1, \mathbb{P}_1, \mathbb{B}_1, \mathbb{V}_1 \rangle$ *and* $\sigma_2 = \langle \mathbb{L}_2, \mathbb{P}_2, \mathbb{B}_2, \mathbb{V}_2 \rangle$ *such that local variables of one state are disjunct from all variables in the other state. Then for a set* $\mathbb{V}$ *of variables*

$$\sigma_1 \diamond_{\mathbb{V}} \sigma_2 ::= \langle \mathbb{L}_1 \uplus \mathbb{L}_2, \mathbb{P}_1 \uplus \mathbb{P}_2, \mathbb{B}_1 \wedge \mathbb{B}_2, (\mathbb{V}_1 \cup \mathbb{V}_2) \backslash \mathbb{V} \rangle$$

This definition is further lifted to equivalence classes. In that case, the merge operation assumes that two representants with accordingly disjunct variables are selected: $[\sigma_1] \diamond_{\mathbb{V}} [\sigma_2] ::= [\sigma_1 \diamond_{\mathbb{V}} \sigma_2]$

For $\mathbb{V} = \emptyset$, $\sigma_1 \diamond \sigma_2$ *and* $[\sigma_1] \diamond [\sigma_2]$ *is written, respectively.*

Lemma 1 states that equivalence is maintained by the merge operator. The proof is similar to [7][p.50f].

Lemma 1 ($\diamond_{\mathbb{V}}$ maintains Equivalence). *Let $\sigma_1 \equiv_! \sigma_2$, then $(\sigma_1 \diamond_{\mathbb{V}} \tau) \equiv_! (\sigma_2 \diamond_{\mathbb{V}} \tau)$ for all $\mathbb{V}$.*

Proof. W.l.o.g. let $\sigma_i = \langle \mathbb{L}_i, \mathbb{P}_i, \mathbb{B}_i, \mathbb{V}' \rangle$ for $i = 1, 2$ and let $\tau = \langle \mathbb{L}, \mathbb{P}, \mathbb{B}, \mathbb{V}'' \rangle$ such that the variables are disjunct according to Definition 8. Let $\bar{y}_1, \bar{y}_2$ be the local variables of σ_1 and σ_2 respectively. According to Theorem 1:

$$CT \models \forall(\mathbb{B}_1 \rightarrow \exists \bar{y}_2.((((\mathbb{L}_1 \triangle \mathbb{L}_2) \uplus \mathbb{P}_1) \bowtie \mathbb{P}_1) \wedge (\mathbb{P}_1 \bowtie \mathbb{P}_2) \wedge \mathbb{B}_2) \wedge \\ \forall(\mathbb{B}_2 \rightarrow \exists \bar{y}_1.((((\mathbb{L}_1 \triangle \mathbb{L}_2) \uplus \mathbb{P}_1) \bowtie \mathbb{P}_1) \wedge (\mathbb{P}_1 \bowtie \mathbb{P}_2) \wedge \mathbb{B}_1))$$

Let $\bar{x} = (V' \cap V)$, then

$$CT \models \forall(\mathbb{B}_1 \rightarrow \exists \bar{y}_2 \exists \bar{x}.((((\mathbb{L}_1 \triangle \mathbb{L}_2) \uplus \mathbb{P}_1) \bowtie \mathbb{P}_1) \wedge (\mathbb{P}_1 \bowtie \mathbb{P}_2) \wedge \mathbb{B}_2) \wedge \\ \forall(\mathbb{B}_2 \rightarrow \exists \bar{y}_1 \exists \bar{x}.((((\mathbb{L}_1 \triangle \mathbb{L}_2) \uplus \mathbb{P}_1) \bowtie \mathbb{P}_1) \wedge (\mathbb{P}_1 \bowtie \mathbb{P}_2) \wedge \mathbb{B}_1))$$

As $(\mathbb{L} \triangle \mathbb{L})$ is always empty and $(\mathbb{P} \bowtie \mathbb{P})$ is a tautology, $(((\mathbb{L}_1 \triangle \mathbb{L}_2) \uplus \mathbb{P}_1) \bowtie \mathbb{P}_1)$ can be extended to $((((\mathbb{L}_1 \uplus \mathbb{L}) \triangle (\mathbb{L}_2 \mathbb{L})) \uplus (\mathbb{P}_1 \uplus \mathbb{P})) \bowtie (\mathbb{P}_1 \uplus \mathbb{P}))$ and $(\mathbb{P}_1 \bowtie \mathbb{P}_2)$ to $((\mathbb{P}_1 \uplus \mathbb{P}) \bowtie (\mathbb{P}_2 \uplus \mathbb{P}))$. Similarly, $\mathbb{B} \rightarrow \mathbb{B}$ is a tautology, and therefore we have for $\bar{z}$ being the local variables of τ combined with $V'' \backslash V$:

$$CT \models \forall(\mathbb{B}_1 \wedge \mathbb{B} \rightarrow \exists \bar{y}_2 \exists \bar{x} \exists \bar{z}.(((((\mathbb{L}_1 \uplus \mathbb{L}) \triangle (\mathbb{L}_2 \mathbb{L})) \uplus (\mathbb{P}_1 \uplus \mathbb{P})) \bowtie \\ (\mathbb{P}_1 \uplus \mathbb{P})) \wedge ((\mathbb{P}_1 \uplus \mathbb{P}) \bowtie (\mathbb{P}_2 \uplus \mathbb{P})) \wedge \mathbb{B}_2) \wedge \\ \forall(\mathbb{B}_2 \rightarrow \exists \bar{y}_1 \exists \bar{x} \exists \bar{z}.(((((\mathbb{L}_1 \uplus \mathbb{L}) \triangle (\mathbb{L}_2 \mathbb{L})) \uplus (\mathbb{P}_1 \uplus \mathbb{P})) \bowtie (\mathbb{P}_1 \uplus \mathbb{P})) \wedge \\ ((\mathbb{P}_1 \uplus \mathbb{P}) \bowtie (\mathbb{P}_2 \uplus \mathbb{P})) \wedge \mathbb{B}_1))$$

As the local variables of $\sigma_1 \diamond_{\mathbb{V}} \tau$ are $\bar{x} \cup \bar{y}_1 \cup \bar{z}$, and analogously for $\sigma_2 \diamond_{\mathbb{V}} \tau$, it can be concluded by Theorem 1

$$\sigma_1 \diamond_{\mathbb{V}} \tau = \langle \mathbb{L}_1 \uplus \mathbb{L}, \mathbb{P}_1 \uplus \mathbb{P}, \mathbb{B}_1 \wedge \mathbb{B}, (\mathbb{V}' \cup \mathbb{V}'') \backslash \mathbb{V} \rangle \equiv_! \\ \langle \mathbb{L}_2 \uplus \mathbb{L}, \mathbb{P}_2 \uplus \mathbb{P}, \mathbb{B}_2 \wedge \mathbb{B}, (\mathbb{V}' \cup \mathbb{V}'') \backslash \mathbb{V} \rangle = \sigma_2 \diamond_{\mathbb{V}} \tau$$

□

4 Confluence Test

This section describes a confluence test for $\omega_!$. The confluence test works in a similar way as it is described in [3] for ω_{va}. It is shown that the test works and how the persistent constraint store influences it. This is not trivial, since $\omega_!$ breaks with monotonicity which is used in the proof for ω_{va}.

Definition 9 defines joinability of two states. This is needed for the definition of confluence itself [3, p. 102]. Definition 10 and 11 are general CHR definitions and not explicitly only for $\omega_!$.

Definition 9 (Joinability). *Two states σ_1 and σ_2 are joinable if there exists a state σ' such that $[\sigma_1] \mapsto_!^* [\sigma']$ and $[\sigma_2] \mapsto_!^* [\sigma']$.*

Definition 10 defines confluence formally, but is not useful for actual confluence tests since in general there exists an infinite number of states [3, p. 102].

Definition 10 (Confluence). *A CHR program is confluent if for all states* S, S_1, S_2 *if* $S \mapsto^* S_1, S \mapsto^* S_2$ *then* S_1 *and* S_2 *are joinable.*

Definition 11 defines local confluence which is later used in the confluence test for terminating programs [3, p. 104].

Definition 11 (Local Confluence). *A CHR program is locally confluent if for all states* S, S_1, S_2*:* *If* $S \mapsto S_1$, $S \mapsto S_2$ *then* S_1 *and* S_2 *are joinable.*

Definition 11 is useful because according to Newman's Lemma for arbitary reduction systems local confulence and confluence coincide for terminating programs [3, p. 104].

Lemma 2 (Newman's Lemma). *A terminating reduction system is confluent iff it is locally confluent.*

Since the transition rules of $\omega_!$ are only applied if the resulting state is not equivalent to the original state $\omega_!$ does not have the monotonicity property that is needed in the confluence criterion for ω_{va}. Lemma 3 is a weaker property that can be sufficient in many cases where proofs for other semantics use the monotonicity property. This lemma is needed for the proofs further on.

Lemma 3. *If* $[\sigma] \mapsto_!^* [\tau]$ *then* $[\sigma] \diamond_{\mathbb{V}} [\sigma'] \mapsto_!^* [\tau] \diamond_{\mathbb{V}} [\sigma']$

Proof. by induction:
In the following r_{seq} is a sequence of rules.
Basis: Let r_{seq} consist of 0 rules.
$[\sigma] \mapsto_!^{r_{seq}} [\tau]$
$[\sigma] \diamond_{\mathbb{V}} [\sigma'] \mapsto_!^* [\tau] \diamond_{\mathbb{V}} [\sigma']$ is correct since σ and τ are equivalent.

Induction hypothesis: if r_{seq} consists of n rules and $[\sigma] \mapsto_!^{r_{seq}} [\tau]$ then $[\sigma] \diamond_{\mathbb{V}} [\sigma'] \mapsto_!^* [\tau] \diamond_{\mathbb{V}} [\sigma']$ is true.

Inductive step: Let r_{seq} consist of $n+1$ rules, r'_{seq} be the sequence of the first n rules of r_{seq} and r_{n+1} be the last rule of r_{seq}. This means according to the induction hypothesis
$[\sigma] \mapsto_!^{r'_{seq}} [\tau]$, $[\sigma] \diamond_{\mathbb{V}} [\sigma'] \mapsto_!^* [\tau] \diamond_{\mathbb{V}} [\sigma']$ and $[\tau] \mapsto_!^{r_{n+1}} [\tau']$
Now $[\tau] \diamond_{\mathbb{V}} [\sigma'] \mapsto_!^* [\tau'] \diamond_{\mathbb{V}} [\sigma']$ needs to be shown.
If r_{n+1} is applicable to $[\tau] \diamond_{\mathbb{V}} [\sigma']$ [7][p.51] proves $[\tau] \diamond_{\mathbb{V}} [\sigma'] \mapsto_!^{r_{n+1}} [\tau'] \diamond_{\mathbb{V}} [\sigma']$.
In the case that r_{n+1} is not applicable to $[\tau] \diamond_{\mathbb{V}} [\sigma']$ then that means that $[\tau] \diamond_{\mathbb{V}} [\sigma'] \equiv_! [\tau'] \diamond_{\mathbb{V}} [\sigma']$ since the head and guard of r_{n+1} are satisfied by $[\tau]$ and so the only way to prevent rule application in $\omega_!$ is if the resulting state is equivalent. □

With lemma 2 there is still an infinite number of states to be tested in general. Definition 12 gives the definition for critical pairs which is used to reduce the number of states that need to be tested to a finite number of states [3, p. 103][7].

Definition 12 (Critical Ancestor State, Critical Pair). *For any two (not necessarily different) rules of a CHR program with renamed apart variables that are of the form*

$$r_1 \ @ \ H_1 \backslash H_2 \ \Leftrightarrow \ G \mid B_c, B_b$$
$$r_2 \ @ \ H_1' \backslash H_2' \ \Leftrightarrow \ G' \mid B_c', B_b'$$

let $O_1 \subseteq H_1, O_2 \subseteq H_2, O_1' \subseteq H_1', O_2' \subseteq H_2'$ *such that for* $B ::= ((O_1 \uplus O_2) = (O_1' \uplus O_2')) \wedge G \wedge G'$ *it holds that* $CT \models \exists . Band(O_2 \uplus O_2') \neq \emptyset$*, then all states of the form*

$$\sigma = \langle L; P; B; \mathbb{V} \rangle$$

where $L \uplus P = K \uplus K' \uplus R \uplus R' \uplus O_1 \uplus O_2$*,* $\mathbb{V}$ *is the set of all variables occurring in heads and guards of both rules and* $K ::= H_1 \backslash O_1, K' ::= H_1' \backslash O_1', R ::= H_2 \backslash O_2, R' ::= H_2' \backslash O_2'$ *are called critical ancestor states. The rules* r_1 *and* r_2 *are called overlapping rules. The pair of states* (σ_1, σ_2) *with*

$$\sigma_1 \ ::== \ \begin{cases} \langle ((K \uplus K' \uplus R' \uplus O_1) \backslash P); P \uplus B_c; B \wedge B_b; \mathbb{V} \rangle & \text{if } H_2 \subseteq P \\ \langle ((K \uplus K' \uplus R' \uplus O_1) \backslash P) \uplus B_c; P; B \wedge B_b; \mathbb{V} \rangle & \text{else} \end{cases}$$

$$\sigma_2 \ ::== \ \begin{cases} \langle ((K \uplus K' \uplus R \uplus O_1') \backslash P); P \uplus B_c'; B \wedge B_b'; \mathbb{V} \rangle & \text{if } H_2' \subseteq P \\ \langle ((K \uplus K' \uplus R \uplus O_1') \backslash P) \uplus B_c'; P; B \wedge B_b'; \mathbb{V} \rangle & \text{else} \end{cases}$$

is called a critical pair of the critical ancestor state σ

Since $\omega_!$ states have two constraint stores each overlap leads to up to 2^x critical ancestor states.

With the definition of critical pairs the actual confluence test is presented in theorem 2.

Theorem 2. *A terminating* $\omega_!$ *program is confluent iff all its critical pairs are joinable.*

Proof. Because of Newman's Lemma 2 it is sufficient to prove local confluence.

To show the if direction, let σ be an $\omega_!$ state where at least two transitions are possible.

$$\sigma \mapsto_! \sigma_1 \text{ and } S \mapsto_! \sigma_2$$

If the two rules apply to different parts of the state σ then σ_1 and σ_2 must be joinable due to lemma 3.

Else the two rules overlap. There must exist critical pair (σ_1', σ_2') with the same overlap and a state σ_{rest} so that $\sigma_1' \diamond \sigma_{rest} \equiv_! \sigma_1$ and $\sigma_2' \diamond \sigma_{rest} \equiv_! \sigma_2$, as a consequence of lemma 3 this means that if (σ_1', σ_2') is joinable then σ_1 and σ_2 are also joinable.

The only if direction can be shown by contradiction. Let P be a program that is locally confluent in $\omega_!$ and has a critical pair that is not joinable. The critical ancestor state of this critical pair can be constructed as initial state. So there exists a state that leads to the nonjoinable critical pair, but since P is locally confluent, the states must be joinable. This results in a contradiction. □

Theorem 2 gives a decidable test for confluence in $\omega_!$. The number of test cases is exponentially increased in comparison to the confluence test for ω_{va}

Example 3. Consider the following program for transitive closure [3][p.190]:

$$dp \ @ \ p(X,Y) \setminus p(X,Y) \Leftrightarrow true.$$
$$p1 \ @ \ e(X,Y) \Rightarrow p(X,Y).$$
$$p2 \ @ \ p(X,Y), p(Y,Z) \Rightarrow p(X,Z).$$

This program is written for semantics that rely on a deterministic order in which rules are executed. The rule dp removes duplicates of $p/2$ constraints. This needs to happen before $p2$ is executed otherwise this program would not be guaranteed to terminate. In $\omega_!$ however it terminates for all possible inputs.

This program has several critical pairs that result from the overlap of dp and $p2$. Here are two of the 12 critical ancestor states:

$$\sigma_1 = \langle \{p(X,X), p(X,X)\}, \emptyset, \emptyset, \{X\} \rangle$$
$$\sigma_2 = \langle \{p(X,X), p(X,X), p(X,X)\}, \emptyset, \emptyset, \{X\} \rangle$$

Each critical ancestor state leads to two equivalent critical pairs, so only two critical pairs need to be tested. For better readability states are represented by their logical reading and persistent constraints are marked with the index p. Critical pair resulting from $\langle \{p(X,X), p(X,X)\}, \emptyset, \emptyset, \{X\} \rangle$:

$$\sigma_1 \mapsto_!^{dp} p(X,X) \mapsto_!^{p2} \underline{p(X,X), p_p(X,X)}$$
$$\sigma_1 \mapsto_!^{p2} p(X,X), p(X,X), p_p(X,X) \mapsto_!^{dp} \underline{p(X,X), p_p(X,X)}$$

Critical pair resulting from $\langle \{p(X,X), p(X,X), p(X,X)\}, \emptyset, \emptyset, \{X\} \rangle$:

$$\sigma_2 \mapsto_!^{dp} p(X,X), p(X,X) \mapsto_!^{p2} p(X,X), p(X,X), p_p(X,X) \mapsto_!^{dp} \underline{p(X,X), p_p(X,X)}$$
$$\sigma_2 \mapsto_!^{p2} p(X,X), p(X,X), p(X,X), p_p(X,X) \mapsto_!^{dp} p(X,X), p(X,X), p_p(X,X)$$
$$\mapsto_!^{dp} \underline{p(X,X), p_p(X,X)}$$

Both critical pairs are joinable. The rest of the critical ancestor states are analog and also lead to joinable critical pairs, so the program is confluent for $\omega_!$. It can be noted that if the input consists only of $e/2$ constraints, then the dp rule is unnecessary since all $p/2$ constraints are added to the persistent store and the semantics of $\omega_!$ would already prevent duplicates.

5 Confluence Checker

The theoretical results of the previous section are used to create a confluence checker for $\omega_!$. It can test syntactical correct programs for confluence. Those programs are not allowed to contain rules that recreate removed head constraints, so called pathological rules. It only supports the Prolog unification = /2 and

`true` as built-ins. It has the option to also support the built-ins `=</2`, `>=/2`, `</2`, `>/2` and `==/2`. These are implemented as constraint solvers with `ask` and `entailed` constraints to replace the built-ins in the guards and can be extended for further built-ins, as it is presented in [9]. For the joinability test it runs a source to source transformation that implements $\omega_!$ and checks if the final states are equivalent.

The application is based on the confluence checker for the abstract semantics that has been written in 2010 by Johannes Langbein [6].

5.1 Modifying the Confluence Checker

The idea behind the modification for $\omega_!$ is to use the source to source transformation of [2] to create a transformed program. The overlaps that are found in the untransformed program are then used to create critical pairs that can be tested with the transformed program in the joinability test.

The confluence checker starts by parsing all rules to search for overlaps. Since during the parsing process all of the necessary information for the source to source transformation is present, it is also used to create the transformed program according to Definition 13. This transformation is based on the implementation of $\omega_!$ as source to source transformation that is presented in [2]. It creates a transformed program that behaves like the original program would behave in $\omega_!$. This transformed program is executed in the so-called refined semantics [3] that is typically used in implementations. In this semantics the order of the rules has influence on the priority of their execution. To differentiate between linear and persistent constraints, each constraint gets an additional argument. This is either l for linear constraints, p for persistent constraints, t for potentially added linear constraints or c for potentially added persistent constraints.

Definition 13 (Source to Source Implementation). *For every n-ary constraint c/n in P there exists a constraint $c/(n+1)$ in $[\![P]\!]$. In the following, for a multiset of user-defined $\omega_!$ constraints $M = \{c_1(\bar{t}_1), ..., c_n(\bar{t}_n)\}$ let $l(M) = \{c_1(l, \bar{t}_1), ..., c_n(l, \bar{t}_n)\}$, $p(M) = \{c_1(p, \bar{t}_1), ..., c_n(p, \bar{t}_n)\}$,*
$t(M) = \{c_1(t, \bar{t}_1), ..., c_n(t, \bar{t}_n)\}$ and $c(M) = \{c_1(c, \bar{t}_1), ..., c_n(c, \bar{t}_n)\}$.
The following source to source transformation shows how the rules of $[\![P]\!]$ are created [2].

1. *For every rule r @ $H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b$ in P, and all multisets $H_1^l, H_1^p, H_2^l, H_2^p$ s.t. $H_1^l \uplus H_1^p = H_1$ and $H_2^l \uplus H_2^p = H_2$ and $H_2^l \neq \emptyset$, the following rule is added at the end of $[\![P]\!]$:*

$$l(H_1^l) \uplus p(H_1^p) \uplus p(H_2^p) \setminus t(H_2^l) \Leftrightarrow G \mid t(B_c), B_b$$

2. *For every rule r @ $H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b$ in P, and all multisets $H_1^l, H_1^p, H_2^l, H_2^p$ s.t. $H_1^l \uplus H_1^p = H_1$ the following rule is added at the end of $[\![P]\!]$:*

$$l(H_1^l) \uplus p(H_1^p) \uplus p(H_2) \Rightarrow G \mid c(B_c), B_b$$

3. *For every rule* $\{c(p,\bar{t}), c(p,\bar{t}')\} \uplus H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b$ *in* $[\![P]\!]$ *with fresh variables where* $\bar{t}$ *and* $\bar{t}'$ *are unifiable, add also the following rule at the end of* $[\![P]\!]$ *with* $\bar{t} \doteq \bar{t}'$*:*

$$\{c(p,\bar{t})\} \uplus H_1 \setminus H_2 \Leftrightarrow G \mid B_c, B_b$$

4. *For every user-defined constraint* c/n *in* P*, add the following rules at the top of* $[\![P]\!]$*, where* $\bar{t}$ *is a sequence of* n *different variables:*

$$\begin{array}{ll} c(p,\bar{t}) \backslash c(c,\bar{t}) \Leftrightarrow \top & c(p,\bar{t}) \backslash c(t,\bar{t}) \Leftrightarrow \top \\ c(c,\bar{t}) \Leftrightarrow c(p,\bar{t}) & c(t,\bar{t}) \Leftrightarrow c(l,\bar{t}) \\ c(p,\bar{t}) \backslash c(p,\bar{t}) \Leftrightarrow \top & c(p,\bar{t}) \backslash c(l,\bar{t}) \Leftrightarrow \top \end{array}$$

Unlike in [2] this transformation is not designed for the priority based semantics. Instead it uses the refined semantics [3] that is used for the CHR implementation in SWI Prolog. In this semantics propagation rules can only fire once with the same constellation of constraints and the priority for rule execution is from top to bottom. Additionally rule 3 now performs the equivalence check of arguments by matching the arguments of the head constraints. The original definition uses an equivalence check in the guard, which would not work in the intended way for free variables in SWI Prolog. Rule 4 is extended for the new equivalence definition and adds a cleanup rule for persistent constraints. This ensures that the final state has the smallest number of constraints possible for easy state equivalence checks.

The transformed program can be executed in SWI Prolog and behaves like the untransformed program would be behave in $\omega_!$ [2]. To differentiate between linear and persistent constraints, each constraint has an additional argument. So only one goal store for CHR constraints is needed in a state. An initial query and its final state of the execution in the transformed program can be represented as terms with the form $state(G, B, V)$. This is the representation needed for the state equivalence checker. The states are always reduced to the equivalent states with the smallest number of constraints according to condition 5 of Definition 4 and Definition 6. This allows the equivalence checker to be used for final states. The parser can be used on its own as an implementation of $\omega_!$ as source to source transformation.

The confluence check is modified so that for every found overlap all possible critical pairs are created according to definition 12. Each constraint of a critical pair has the additional argument to determine if the constraint are added as linear or persistent.

Example 4. Calling the confluence checker with the following program.

```
b <=> true.
a <=> true.
a ==> b.
```

This program is confluent in the abstract semantics, but not in $\omega_!$ due to the non removability of persistent constraints. The checker finds a non-joinable critical pair and gives the following output:

```
======================================================================
The following critical pair is not joinable:
state([],[true],[])
state([a(1),b(c)],[],[])
This critical pair stems from the critical ancestor state:
[a]
with the overlapping part:
[(a,a)]
of the following two rules:
a<=>true
a==>b
======================================================================
'The CHR program in ''E:\\examples\\example-1.pl'' is NOT confluent!
'1' non-joinable critical pair(s) found!'
```

5.2 Limitations

The source to source transformation that is used to simulate the behavior of $\omega_!$ cannot be used with programs that contain pathological rules. Programs that have pathological rules need to be rewritten by splitting the rule in several rules with appropriate guards, where constraints are added to the kept head instead of removing and adding them. Since this also removes potential trivial non-termination in $\omega_!$, the resulting program is not operationally equivalent to the original program, but usually has the intended behavior.

The source to source transformation creates many rules for a single rule in the original program. The worst case of the blowup of rules for a single rule lies in $\mathcal{O}(2^n n!)$ where n is the number of head constraints of the rule, while the best case still lies in $\mathcal{O}(2^n)$. There can be redundant rules or propagation rules with `true` as body. The later kind of rule causes the confluence checker to write compiler warnings to the console during the joinability tests. These problems could be fixed by a real implementation of $\omega_!$. If such an implementation is made, the source to source transformation can be removed from the confluence checker and it can be modified to use the real implementation for the confluence tests without changing the overall operation of the confluence checker.

The confluence checker only supports a limited number of built-ins. While more built-ins can be added by constraint solvers those can still be limited, especially when trying to add something like mathematical operations e.g. addition.

Since the joinability test only checks final states for equivalence some joinable critical pairs may be presented as non-joinable. If this happens the non-joinability that leads to different final states is implied by another critical pair that is not joinable.

6 Related Work

In [1] the idea of persistent constraints is introduced, $\omega_!$ is defined and its termination behavior is analyzed. The equivalence definition in this work misses the idea that linear constraints can be implied by persistent constraints in the context of state equivalence. The implementation of $\omega_!$ as source to source transformation is presented in [2]. It has an insufficient definition for pathological rules and had to be adjusted for the new equivalence definition. In [7] a decidable criterion for equivalence in $\omega_!$ is introduced. This is for the original state equivalence definition and had to be adjusted for the new definition. Additionally [7] presents a lemma for monotonicity in $\omega_!$ which is incorrect. Our work introduced a weaker property to replace the incorrect monotonicity lemma. So far there were no analysis of confluence behavior and confluence tests for $\omega_!$.

In [6] the confluence checker for the abstract semantics is presented. The test for confluence that it uses is described by [3]. This is used as foundation for the confluence checker for $\omega_!$ and extended by built-ins that are not supported in the original implementation.

7 Conclusion and Future Work

A confluence test for the operational semantics $\omega_!$ for CHR with persistent constraints has been presented (c.f. theorem 2). For this purpose, the state equivalence definition of $\omega_!$ states has been adapted such that it better reflects the intuitive meaning of persistent constraints. Other definitions and results for the original definition have been shown to be compatible with the improved state equivalence definition (c.f. section 3).

The standard confluence test of CHR has been adapted to match $\omega_!$. As it heavily relies on monotonicity which is broken by $\omega_!$, a weaker property has been established to allow for the confluence test (c.f. lemma 3). The number of critical pairs increases exponentially compared to ω_{va} (c.f. definition 12).

The confluence test has been implemented based on the confluence checker for the abstract semantics of CHR [6] (c.f. section 5). Due to the close relation of the proposed confluence criterion to the criterion for ω_{va}, the confluence checker for $\omega_!$ has less limitations than the confluence checker for the abstract semantics on which it is based. The confluence checker integrates support for built-in constraints other than $\mathit{true}, \mathit{false}$ and syntactic equality $=$. The general interface of this extension allows for integration of even more built-in constraints.

For the future it can be investigated if confluence in $\omega_!$ also implies the linear logical reading of programs and easy parallelization. Since most existing CHR programs are written with the abstract semantics with token store in mind, many programs have to be modified to work properly in $\omega_!$. Hence, it would be interesting to investigate programs that have been originally been written for $\omega_!$ and their behavior regarding parallelization. Furthermore, a direct implementation of $\omega_!$ that replaces the source to source transformation in the confluence checker would be useful as it would improve efficiency of the execution of the programs

and the confluence checker. The results presented in this paper are independent of the actual implementation of $\omega_!$ and could therefore be used without deviation together with such a direct implementation.

Further it would be interesting to investigate so-called completion methods for $\omega_!$. Those are methods that add rules to a non confluent program in order to make it confluent, like it is investigated in [4] for CHR without persistent constraints.

To improve the efficiency of the confluence test for $\omega_!$ it should be investigated if the number of critical pairs can be reduced.

Acknowledgments

We want to thank the reviewers for their detailed feedback which was a great help for improving this work.

References

1. Betz, Hariolf ; Raiser, Frank ; Frühwirth, Thom: Persistent constraints in constraint handling rules. In: *WLP* 9 (2010), S. 155–166
2. Betz, Hariolf ; Raiser, Frank ; Frühwirth, Thom: A complete and terminating execution model for constraint handling rules. In: *Theory and Practice of Logic Programming* 10 (2010), Nr. 4-6, S. 597–610
3. Frühwirth, Thom: *Constraint handling rules*. Cambridge University Press, 2009
4. Abdennadher, Slim ; Frühwirth, Thom: On completion of constraint handling rules. In: *Principles and Practice of Constraint ProgrammingCP98* (1998), S. 25–39
5. De Koninck, Leslie ; Schrijvers, Tom ; Demoen, Bart: User-definable rule priorities for CHR. In: *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming* ACM, 2007, S. 25–36
6. Langbein, Johannes ; Raiser, Frank ; Frühwirth, Thom: A state equivalence and confluence checker for CHR. In: *Proceedings of the 7th International Workshop on Constraint Handling Rules. Report CW* Bd. 588, 2010, S. 1–8
7. Raiser, Frank: *Graph transformation systems in Constraint Handling Rules: improved methods for program analysis*, Universität Ulm, Diss., 2010 `http://dx.doi.org/10.18725/OPARU-1742`
8. Raiser, Frank ; Betz, Hariolf ; Frühwirth, Thom: Equivalence of CHR states revisited. In: *6th International Workshop on Constraint Handling Rules (CHR)*, 2009, S. 34–48
9. Richter, Frank: An Operational Equivalence Checker for CHR. Bachelor Thesis, Ulm University, 2014
10. Richter, Frank: Confluence for Constraint Handling Rules with Persistent Constraints. Master Thesis, Ulm University, 2017 `https://www.informatik.uni-ulm.de/pm/fileadmin/pm/home/fruehwirth/Bachelor-Thesis-Frank-Richter.pdfl`, . – Accessed: 2017-07-25